Rapport de stage

Développement d'une plateforme de tests automatisés : GreenT

Antoine de Roquemaurel

M1 Informatique – Développement Logiciel
2014 – 2015

Maître de stage :
Alain Fernandez

Tuteur universitaire :
Bernard Cherbonneau

Du 04 Mai au 28 Août 2015
Version du 18 août 2015

Je tiens à remercier toutes les personnes m'ayant permis de réaliser ce stage.

En premier lieu, un grand merci à Corinne Tarin pour m'avoir accepté au sein de son équipe.
Je remercie particulièrement Alain Fernandez pour m'avoir suivi et conseillé tout au long de ce stage tout en partageant son expérience.
Une pensée à Joelle Decol pour sa bonne humeur quotidienne, ainsi qu'à toute l'équipe du troisième étage, grâce à qui j'ai passé d'excellents moments au sein de l'entreprise.
Merci à mon tuteur universitaire Bernard Cherbonneau pour son suivi et sa visite en entreprise.
Enfin, je remercie toutes les personnes m'ayant entouré durant ce stage et aidé à la rédaction ce rapport, à savoir Diane, Ophélie, Clément et Mathieu.

Table des matières

4 *GreenT* : fonctionnement général

Comme nous l'avons montré dans le chapitre 3, l'entreprise a besoin d'un nouvel outil aidant aux tests d'intégration : *GreenT*.

Nous allons donc voir le développement et la conception de cette plateforme de tests.

Sommaire

Au début de mon stage, le projet ayant un an, les fonctionnalités développées ci-dessous étaient déjà faites. Je suis cependant intervenu sur la plupart d'entre elles soit pour des corrections de bugs d'une part, soit à des fin d'améliorations d'autre parts.

Afin de présenter mon travail, que vous trouverez chapitre 2, il est nécessaire de présenter le fonctionnement général de cette plateforme afin d'en avoir une vue d'ensemble.

4.1 Le fichier Walkthrough

Le fichier Walkthrough est un fichier qui sera fourni par la personne en charge des tests, c'est un fichier au format Excel qui contient les informations de chacune des variables à tester. Il contient ainsi un très grand nombre de colonnes, bien que seule une partie de celles-ci nous intéressent. Certaines colonnes ont été remplies par le fournisseur du plugin, d'autres colonnes sont ajoutées dans le seul but de la génération de tests automatiques par *GreenT*. Voici les informations les plus intéressantes :

Nom de la variable Le nom de la variable testée : il existe un nom court et un nom long.

Informations aidant à la conversion des données Certains équipements[1] tel que le *debugger* ne fonctionne qu'avec des valeurs Hexadécimales. À la charge de *GreenT* de convertir ces données vers des valeurs physiques exploitables par le testeur. Ces colonnes contiennent les informations nécessaires au calcul de conversion[2].

Nécessité d'un test automatique Un `GreenTTest` ne sera généré que si la colonne vaut *Yes*.

Statut du test La plateforme éditera automatiquement cette colonne afin de reporter le statut

1. Vous trouverez plus d'informations sur le fonctionnement d'une table de tests section 3.1.1, et dans l'annexe **??**
2. Informations tel que le domaine de définition physique et le domaine de définition hexadécimal

du test [3].

Précondition (cf section 1.3.2) Contient un scénario d'initialisation du *workbench* : tension de départ, démarrage de l'ECU, ...

Scénario de stimulation (cf section 1.3.2) Contient un ou plusieurs scénarios de stimulations

`ExpectedBehavior` (comportement attendu, cf section 1.3.3) Contient une expression évaluant les variables ayant été enregistrées durant la stimulation : *GreenT* devra vérifier que cette expression est correcte à toute instant de la stimulation.

Variable à enregistrer (cf section 1.3.3) Contient les variables devant être enregistrées durant un scénario, en plus des variables présentes dans l'expected behavior. Celles-ci peuvent servir en tant que données contextuelles permettant de mieux cerner le résultat d'un test.

Informations du test (cf section 1.3.5) Plusieurs colonnes telles que la sévérité, le responsable du test, les commentaires, ...

	RB_Variable	Test_InterfaceTy	Test_Conditi	Test_Stimulus	Test_ExpectedBehavior	Test_RecordedVariab	Test_LocalAli	Test_Sevel
2	ACCIntVlv_iSens	IO	SUB_ECU_GO	SCENARIO IO_VACC_control1 SUB_SCE_BENCH_ACC_0_100_0_POURC_Acti veBenchMode_0 END SCENARIO	EVAL(ACCIntVlv_iSens=(v_pwm_cfb_acc_slv / NC_C			Low
3	ACCIntVlv_r	IO	SUB_ECU_GO	SCENARIO wait2s SUB_WAIT_2S END SCENARIO	EVAL(pwm_ducy_acc_slv_cus=ACCIntVlv_r , TOLRES			Low
74	Air_uRawTEGRClrDs	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(Air_uRawTEGRClrDs=0)			Low
100	BrkBstP_uRaw	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(BrkBstP_uRaw=0)			Low
117	Clth_b25PrcSens	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(Clth_b25PrcSens=0)			Low
133	Clth_uAnaRaw	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(Clth_uAnaRaw=0)			Low
330	EGRVlv_uRaw	IO	SUB_ECU_GO	SCENARIO IO_EGRV_control SUB_SCE_EGRV_POS_0_5_0_V END SCENARIO	EVAL(EGRVlv_uRaw=v_egrv_mes[0] *1000, TOLRES			Low
357	EnvP_pSens	IO	SUB_ECU_GO	SCENARIO IO_AMP_control SUB_SCE_AMP_0_5_0_V END SCENARIO	EVAL(EnvP_pSens=amp_mes , TOLRES(1))			Low
362	EnvT_tSens	CALCULATION	SUB_ECU_GO	SCENARIO IO_TAM_control SUB_SCE_TAMB_0_5_0_V END SCENARIO	EVAL(EnvT_tSens=temp_air_mes[2] , TOLRES(1))			Low
363	EnvT_uRaw	IO	SUB_ECU_GO	SCENARIO IO_TAM_control SUB_SCE_TAMB_0_5_0_V END SCENARIO	EVAL(EnvT_uRaw=vp_temp_air[2]*1000 , TOLRES(1)			Low
389	Epm_stEposCPF	CALCULATION	SUB_ECU_GO	SCENARIO CALC_N_500_control SUB_SCE_N_0_500_0_RPM END SCENARIO	IF (lv_first_vld_tooth=0) THEN EVAL(Epm_stEposCPI			Low
422	Exh_uRawTOxiCatDs	IO	SUB_ECU_GO	SCENARIO IO_TEMP_UP_CAT_control SUB_SCE_TEG_UP_CAT_0_0_5_0_V END SCENARIO	EVAL(Exh_uRawTOxiCatDs=vp_teg_sns_avl[2] , TOLI			Low
426	Exh_uRawTPFltDs	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(Exh_uRawTPFltDs=0)			Low
439	FlSys_stDeflate	CALCULATION	SUB_ECU_GO	SCENARIO wait2s SUB_WAIT_2S END SCENARIO	EVAL(FlSys_stDeflate=state_fuel_pipe_ex_air)			Low
441	Fl_uRawFlFWLvl	IO	SUB_ECU_GO	SCENARIO IO_WFS_control SUB_SCE_WFS_0_5_0_V END SCENARIO	EVAL(Fl_uRawFlFWLvl = vp_fuel_warn , TOLRES(1))			Low
464	Fan_rPs	CALCULATION	SUB_ECU_GO	SCENARIO SCE_CFA_MAN_1 SUB_SCE_CFA_MAN_1_0 END SCENARIO	IF (lv_cfa_2=1) THEN EVAL(Fan_rPs=100)ELSE Fl			Low

Figure 4.1 – Aperçu d'un fichier Walkthrough

3. Un test pouvant être Green ou Red mais peut aussi comporter une erreur, tel qu'un problème d'exécution ou de génération, ...

4.2 Fonctionnement Global

Le développement de *GreenT* inclut un certain nombre de fonctionnalités attendues par le client et indispensable à son fonctionnement. D'autres fonctionnalités pourront apparaître plus tard en fonction des besoins.

Les principaux modules sont les suivants, avec leurs interactions schématisées figure 1.2 : dans des objets ovales sont représentés des fichiers, les carrés représentent des modules de la plateforme et les flèches en pointillés un transfert réseau, les couleurs représentent les différents modules de la plateforme.

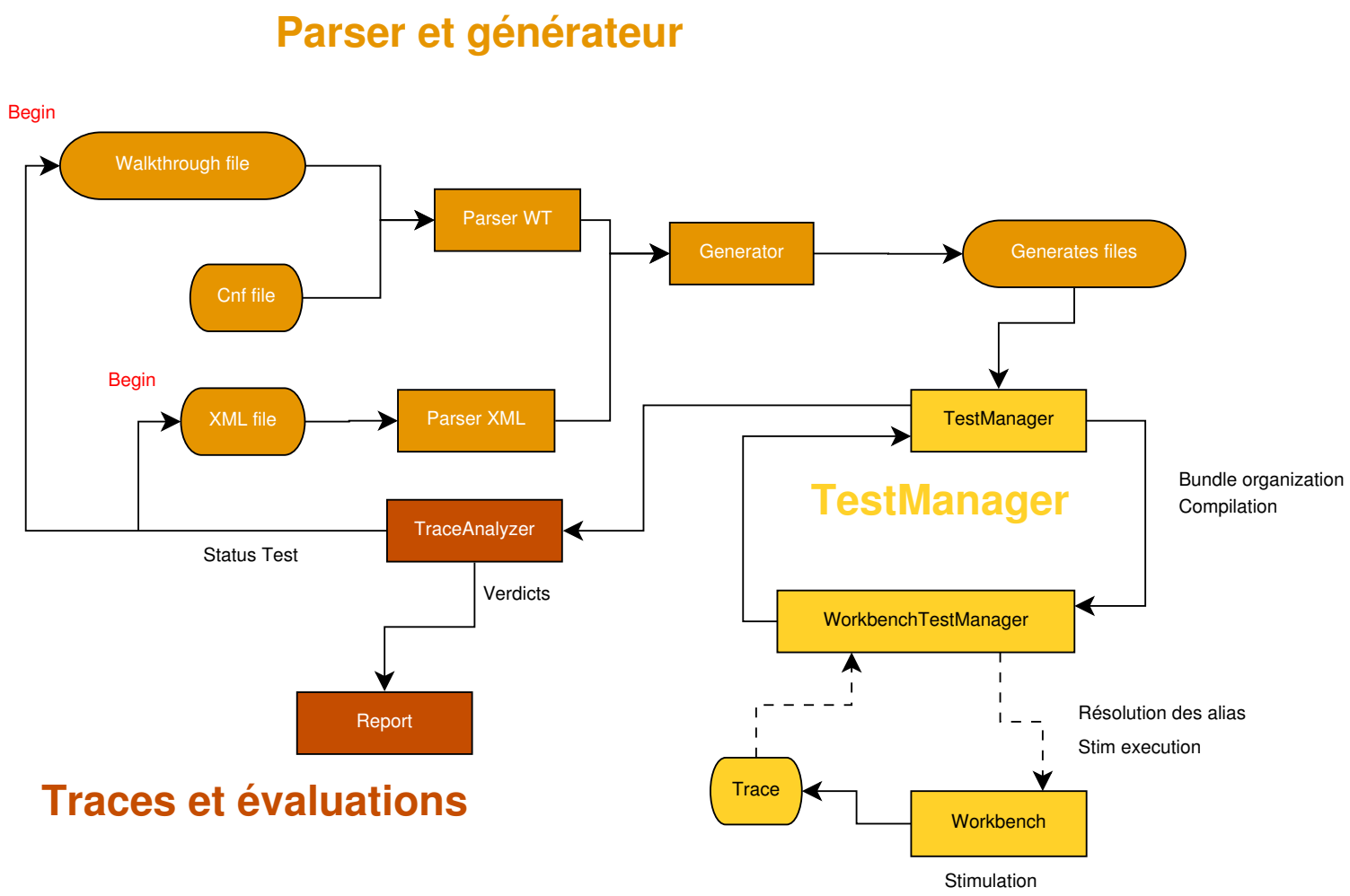


FIGURE 4.2 – Fonctionnement général de la plateforme *GreenT*

Dans la figure 1.2, vous pouvez voir des flèches en pointillés représentant des échanges réseau. En effet l'exécution des stimulations et l'enregistrement des traces se fait par un contrôle distant des bancs de tests [4].

Les principales fonctionnalités demandées par le client sont disponibles sur le client, la majorité peuvent s'effectuer en local :

- Le parsing et la génération (section 1.3.1)
- L'organisation en Bundle (section 1.3.4)

4. La schématisation du fonctionnement d'un banc est disponible section 3 figure 3.1

— La production de rapports détaillés (section 1.3.3)

Alors que d'autres fonctionnalités vont nécessiter la présence de serveurs et de connexion réseau permettant d'effectuer ces actions :
— Les stimulations (section 1.3.2)
— Les enregistrement des traces (section 1.3.2)

4.3 Les fonctionnalités du Client

Afin de répondre au mieux aux attentes du client, il a été choisi d'utiliser le langage Java pour développer notre client, ceci en raison de divers avantages comme le côté multi-plateforme, son typage fort et sa forte communauté qui permet ainsi une maintenance facilitée.

Exemple Afin de comprendre au maximum les tenants et aboutissants de notre plateforme, nous allons utiliser le même exemple tout au long de cette section.

Le testeur souhaite vérifier le bon fonctionnement des ventilateurs de refroidissement du moteur. Nous allons voir de quelle manière il pourrait utiliser *GreenT* afin de tester cela.

4.3.1 Parsing et Génération

Le but premier de la plateforme est d'effectuer des tests automatiques, il est ainsi indispensable d'avoir un système d'automatisation.

Pour cela, nous avons un parser : il analyse un certain type de fichier [5] et en retire pour chaque test, le scénario de pré condition, les différents scénarios de stimulations, leur *Expected Behavior*, les données qui devront être enregistrées ainsi que différentes informations sur le test [6].

Une fois toutes ces données acquises, il les transmet à un générateur qui est en charge d'écrire les fichiers Java de chaque test, tous sont organisés dans un dossier temporaire avec un dossier par test. Le `TestManager` peut ensuite traiter ces données.

Exemple Ci-dessous les différentes cellules qui pourrait être renseigné par le testeur pour notre cas d'étude. Nous verrons ce qu'effectuent précisément ces actions dans la suite de la section.

Précondition	Scénario de Stimulation	Expected Behavior

`// Tension batterie`
`HIL_VB = 13;`
`// Clé moteur`
`HIL_KEY = 1;`
`// Vérifications`
`CHECK(HIL_KEY = 1`
`  && HIL_VB = 13);`
	`// Rampe de vitesse véhicule`
	`// de 0 à 50km/h par pas de 1 toutes les secondes`
	`RAMP(HIL_VS, 0, 50, 1, 1);`
	`TODO vérifier syntaxe de RAMP`
		`if temperature > max then`
		`  -- Ventilateur allumé`
		`  EVAL(cfa_on = 1);`
		`else`
		`  -- Ventilateur éteint`
		`  EVAL(cfa_on = 0);`
		`end if`

5. Nous ne commencerons qu'avec le Walkthrough pour débuter, mais dans le futur nous pourrions avoir des fichiers XML, des bases de données, ...
6. Responsable du test, sévérité, commentaires, nom de la variable, ...

4.3.2 Stimulation

Afin de tester une variable du plugin, les développeurs vont utiliser des alias présents sur un device : actuellement, un HIL ou un debugger, prochainement nous pourrions en utiliser d'autres que ces deux derniers.

Le spécifieur va rédiger des scénarios de stimulation, ceci afin de mettre le contrôleur dans certaines conditions. Son but sera ensuite de vérifier que ces variables restent cohérentes vis-à-vis du scénario effectué.

Un scénario particulier doit être spécifié : une pré condition qui a pour but d'initialiser les *devices* et certains alias afin d'avoir un état de stimulation qui soit cohérent et identique à chaque lancement du scénario. Ce scénario sera effectué avant le lancement de chacun des scénarios de stimulation.

Durant l'exécution d'une stimulation, les variables nécessaires sont enregistrées afin de pouvoir produire des rapports et des verdicts ensuite.

Exemple Comme nous l'avons vu section 1.3.1, le testeur nous a donné un scénario de stimulation ainsi qu'un scénario de précondition. Le code généré va ainsi communiquer avec le serveur HIL afin qu'il envoie les bons stimulis à l'ECU.

Ainsi, comme mis en commentaires, le scénario de précondition va mettre la batterie à 13 Volts, et mettre la clé, nous allons ensuite vérifier que cette action a bien été fait. Si tel est le cas, le scénario de stimulation va être effectué. Ce scénario va effectuer une rampe de vitesse, notre véhicule va aller de zéro à cinquante kilomètre-heure avant de retourner à l'arrêt.

4.3.3 Les traces et leurs évaluations

Lorsqu'un scénario de stimulation s'exécute, un certain nombre de variables sont enregistrées : ces variables sont stockées sous la forme d'une trace au format CSV, qui pourra plus tard être représentée sous forme de courbe.

Une fois que la trace est complète, il est nécessaire de l'évaluer : le spécifieur a décrit le comportement attendu dans la colonne *Expected Behavior* détaillant dans quel cas le test est correct. Cette expression va être transformée en arbre logique afin de l'évaluer à tout instant de la trace.

Exemple Durant notre rampe véhicule, le *debugger* a enregistré différentes variables, en l'occurence nous avons enregistré les trois variables présentes dans notre *Expected Behavior* : la température du moteur – `temperature`, la température maximum sans ventilateur – `max`, ainsi que l'état de notre ventilateur – `cfa_on`.

Ces variables ont été enregistrées durant l'intégralité de la rampe véhicule qui à eu une durée de cinquante secondes. À la fin de la stimulation, le serveur retourne ainsi une trace de cinquante secondes contenant toutes les variables. *GreenT* peut ensuite évaluer notre *expected behavior* sur l'intégralité de l'enregistrement.

4.3.4 Le module TestManager

Le `TestManager` est le chef d'orchestre de *GreenT*, il a donc un certain nombre de responsabilités.

Il va d'abord organiser les différents tests en un concept que nous avons appelé *Bundle* : afin de limiter le temps d'exécution qui atteindra plusieurs dizaines d'heures, il est intéressant de regrouper les tests possédant les mêmes scénarios de stimulations et les mêmes pré conditions. Seules leur *Expected Behavior* changent, mais celles-ci pourront être évaluées sur la même Trace.

Une fois les tests organisés en *Bundle*, il va les compiler et les donner à un `WorkbenchManager` : toujours pour une raison d'optimisation, il sera intéressant de pouvoir exécuter les enregistrements sur plusieurs bancs simultanément, pour cela le `TestManager` sera capable de savoir quels bancs peuvent être utilisés et pourra distribuer ses *bundles* en fonction.

Chaque `WorkbenchManager` sera en charge d'exécuter le code généré plus tôt et dialoguera en réseau avec son banc, une fois l'exécution terminée, il obtiendra une trace qui pourra être évaluée.

Afin d'être le plus souple possible, il existe plusieurs modes d'exécution du `TestManager` :

Check only Essaye de parser les différents fichiers, et vérifie que ceux-ci ne comportent aucune erreur de grammaire, d'alias introuvable, d'écriture sur un alias en lecture seule etc...

Parse and generate bundles Parse les fichiers et génère des jars exécutables répartis en bundle

Parse and execute Parse les fichiers, génère les jars pour les bundles et les exécute : c'est le mode « classique ».

Restart test execution Redémarre une exécution qui se serait mal terminée.

4.3.5 Production de rapport détaillé

La plateforme a en charge la production d'un rapport détaillé pour chaque test. Ce rapport contiendra un certain nombre d'informations, et permettra au testeur de comprendre pourquoi le test n'est pas passé. Voici les informations que contiendra ce rapport :

— Nom du test, de la variable à tester
— Nom du responsable du test
— Sévérité du test
— Pourcentage de branches de l'expectedBehavior renvoyant faux(Test « Rouge »), n'ayant pas pu être testé(Test « Gris ») et étant correct(Test Vert)
— Le testeur aura à sa disposition les expressions concernées par un résultat Rouge ou Gris.
— Les colonnes utiles du `Walkthrough`

Actuellement, les rapports se font au format Excel avec l'intégralité de notre enregistrement et pour chaque timestamp, un verdict. Un exemple de rapport est accessible en Annexe **??** page **??**.

Dans un futur proche, ces rapports pourraient être générés dans un format Web avec une possibilité de naviguer entre plusieurs tests, et d'avoir un affichage des courbes de manière graphique.

4.3.6 Mise à jour du Walkthrough

Une fois l'analyse d'un test exécuté, un résultat est mis dans le fichier Excel, en fonction de l'analyse de la trace : un verdict rouge renverra un résultat rouge, si tout est vert le résultat sera vert et enfin, celui-ci pourra être gris si nous n'avons pas été capable d'évaluer de résultats.

Si il y a eu une erreur à la génération [a] ou à l'exécution [b], la plateforme doit remettre le message d'erreur clair au niveau du test afin que l'utilisateur soit conscient du fait que le test n'a pas pu être exécuté, et pour quelle raison. Libre à lui de corriger le test si nécessaire, ou de le signaler si cela semble être un bogue.

a. Mauvaise syntaxe, variables inexistantes, ...
b. Problèmes réseaux, variable non trouvée, communication entre l'ECU et le debugger, ...

4.4 Le fonctionnalités des serveurs

Comme expliqué précédemment, GreenT va avoir en charge l'exécution de stimulation. Ces stimulations vont communiquer avec une table de tests. Actuellement une table est composée de deux *devices* différents, comportant chacun leur serveur :

— Un HIL, Hardware In the Loop, simulateur d'environnement véhicule
— Un Debugger permettant de voir l'état du programme présent dans l'ECU

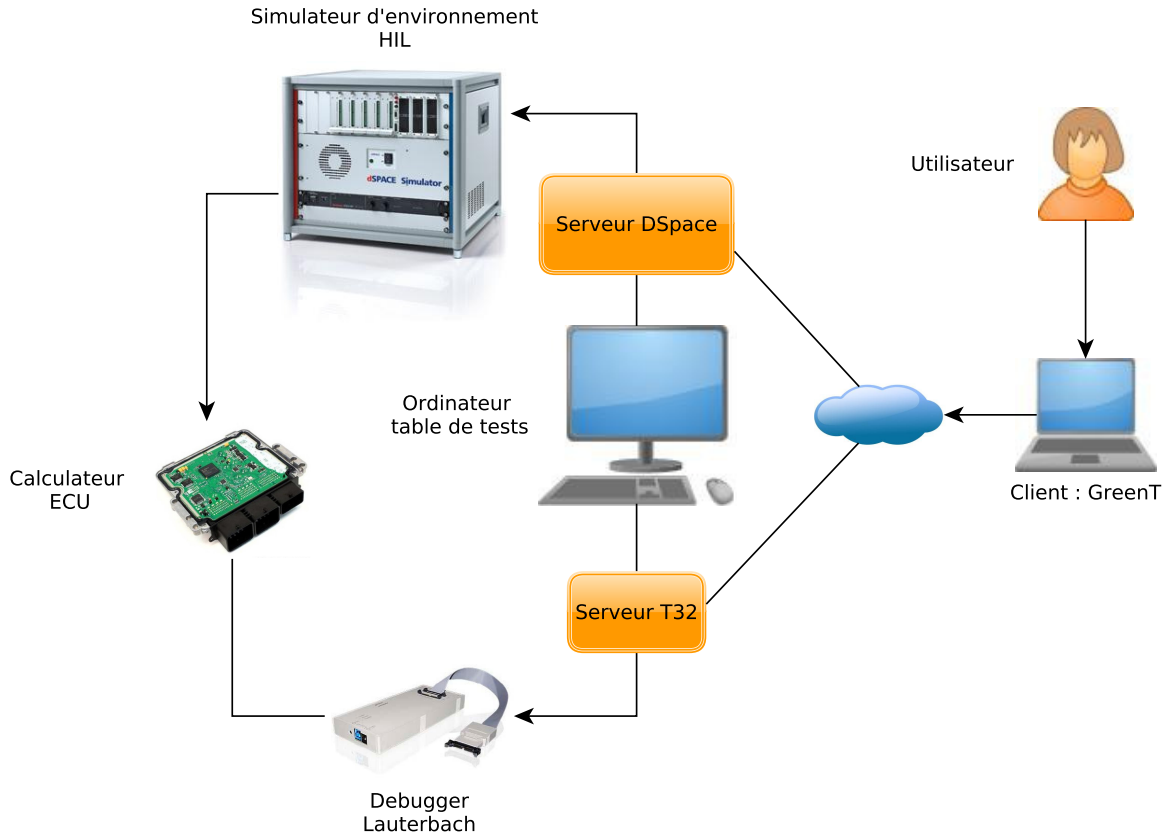


FIGURE 4.3 – Communications de la plateforme

Au début du développement de la plateforme, il a été décidé que les serveurs devront être le plus simple possible pour plusieurs raisons :

— Rabattre le maximum de fonctions métier près du client pour centraliser au maximum le fonctionnement et éviter la maintenance superflue
— Avoir la possibilité de réutiliser les serveurs pour d'autres projets
— Pouvoir ajouter facilement un nouveau device, qui ne nécessiterai que l'ajout d'un nouveau serveur relativement simple

4.4.1 Le serveur Debugger, contrôle de Trace32

Le serveur Debugger propose des services basiques permettant de répondre aux besoins :
- Flasher le logiciel dans la flash de l'ECU
- Démarrer l'ECU
- Arrêter l'ECU
- Lire une variable ou une calibration
- Modifier une variable
- Enregistrer des variables

Afin d'effectuer ces actions, le serveur s'appuie sur une API fournie par l'outil Trace32 permettant de contrôler le debugger. Ainsi tous nos services vont s'appuyer sur cette API. C'est pour cette raison que ce serveur est développé en Java afin de pouvoir utiliser facilement ces fonctions.

4.4.2 Le serveur HIL, contrôle du DSpace

Le serveur DSpace va devoir lui aussi répondre aux différentes stimulations, ainsi ces services sont relativement similaire :
- Modifier une valeur de la base de données
- Lire une valeur de la base de données
- Enregistrer des valeurs

Ce serveur a été développé en réutilisant une partie de ce qui avait été fait pour la TA3, présenté section 3.1.3 afin de ne pas « réinventer la roue ».

À l'instar du serveur Debugger, nous utilisons une API fournie par l'outil permettant de contrôler le HIL : ControlDesk. C'est ainsi que ce serveur est développé en Python afin de répondre à cette contraintes : l'API du ControlDesk est en Python.

5 Ma collaboration au projet

Après avoir défini plus en détails les besoins de notre plateforme et son fonctionnement général, nous allons maintenant voir en détail de quelle manière j'ai contribué à ce projet. En parallèle de la maintenance de notre plateforme, j'ai développé deux nouvelles fonctionnalités. Ces deux fonctionnalités ayant un rapport direct avec la notion de « calibration », nous allons tout d'abord définir celles-ci avant de voir en détail mon développement et la maintenance que j'ai effectué.

Sommaire

5.1 Les calibrations

Une calibration est une constante stockée en flash, c'est-à-dire en mémoire non volatile. Ainsi le logiciel du contrôle moteur peut accéder à toutes ces calibrations en lecture uniquement.

Ces calibrations permettent de configurer un véhicule avant sa mise en production. Cette configuration peut se faire en fonction de plusieurs critères :

- Le matériel en face du calculateur, comme le nombre d'injecteurs
- La version du logiciel du contrôle moteur
- Leur modification permet de faire une mise au point, permettant d'améliorer la consommation par exemple.

Une fois le logiciel d'un calculateur mis en production, ces calibrations ne doivent pas évoluer, les modifier ne sert donc qu'à la mise au point et à la généricité du logiciel.

5.2 Le « patch calib »

Pour un certain nombre de scénario de stimulation, il est nécessaire de modifier des valeurs de calibrations. Cette action demande d'ajouter une grammaire spécifique : `PATCH_CAL(calibration, valeur)`. En interne, cela générera du code permettant de flasher la nouvelle valeur.

5.2.1 Expression du besoin

Les cas d'utilisation

Comme nous pouvons le voir figure 2.1, le patch de calibrations est répertorié en deux grandes parties : le parser et la génération des instructions permettant de changer la valeur d'une calibration, et l'exécution du patch proprement dit.

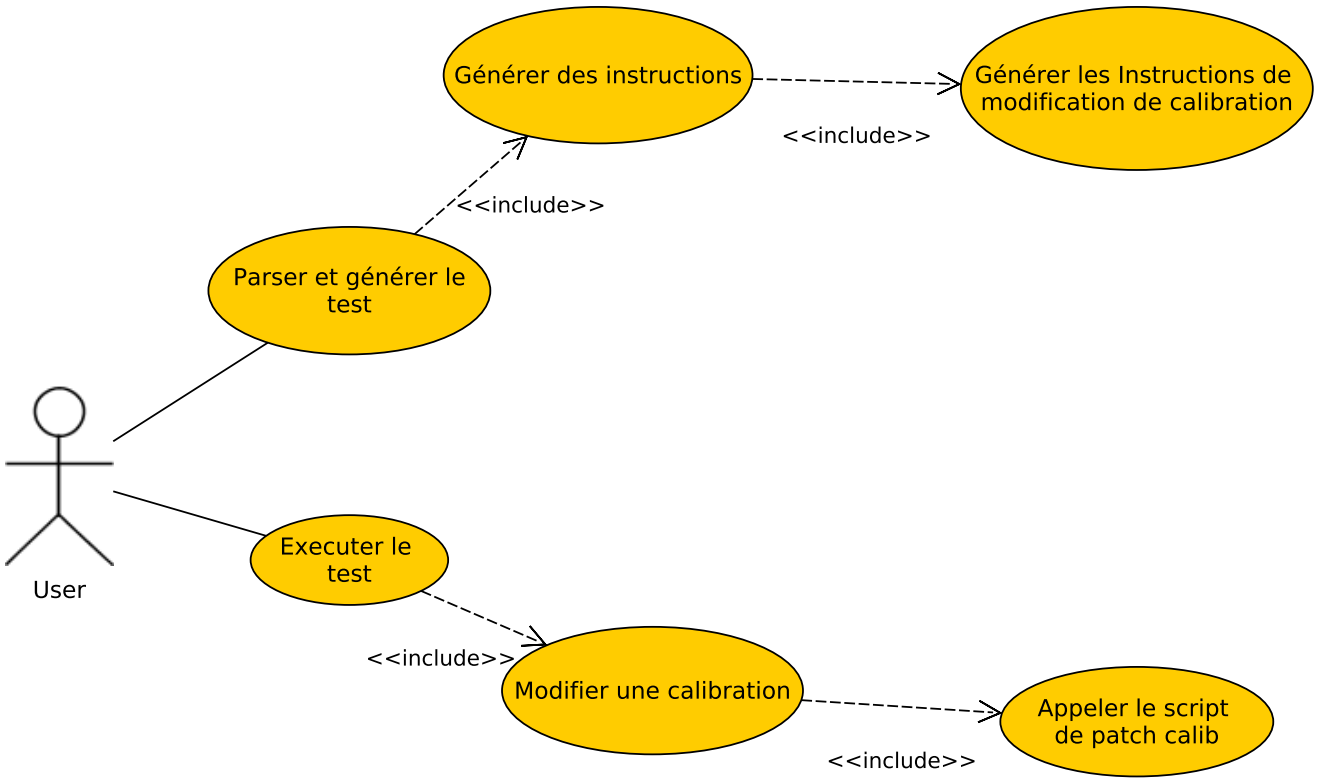


Figure 5.1 – Diagramme de cas d'utilisations du patch de calibrations

Limitations du patch calib

Afin de concevoir au mieux la solution répondant aux besoins du client, j'ai tout d'abord du me renseigner sur la manière dont je pouvais effectuer cette action sur l'ECU, c'est-à-dire l'implémentation du serveur. Après discussions avec les personnes compétentes, une restriction impactant mon développement a été mis au clair.

Afin de modifier une calibration, il est nécessaire de modifier la mémoire flash. Cette modification va nous obliger à éteindre l'ECU, la modification de la flash étant impossible ECU on. Il ne sera donc pas possible de modifier une calibration au milieu d'un scénario de stimulation, le scénario n'aurait aucun sens si nous éteignons et rallumons l'ECU pendant celui-ci.

Cette restriction n'a cependant pas d'impact sur les tests qui seront rédigés, un scénario de stimulation a pour but de se rapprocher du fonctionnement nominal d'une voiture, or il est inconcevable de modifier une calibration pendant que la voiture tourne. Du point de vue des tests, ce choix reste cohérent.

5.2.2 Fonctionnement à l'utilisation

Comme vu section 2.2.1, le « patch calib » ne peut être utilisé que si l'ECU est arrêté. Afin de forcer ce fonctionnement, et d'éviter une mauvaise utilisation, il a été choisi d'utiliser la grammaire du langage. En effet, une mauvaise utilisation de la fonctionnalité nous renverra une *syntax error*, forçant l'utilisateur à corriger cela.

```
SCENARIO NomDuScenario
  PATCH SECTION
    patch_cal(cal1, value1);
    patch_cal(cal2, value2);
  END SECTION
  // Stimulation proprement dite
  // Rampe, wait, ...
END SCENARIO
```

Extrait de code 5.1 – Scénario de stimulation contenant des patchs de calibration

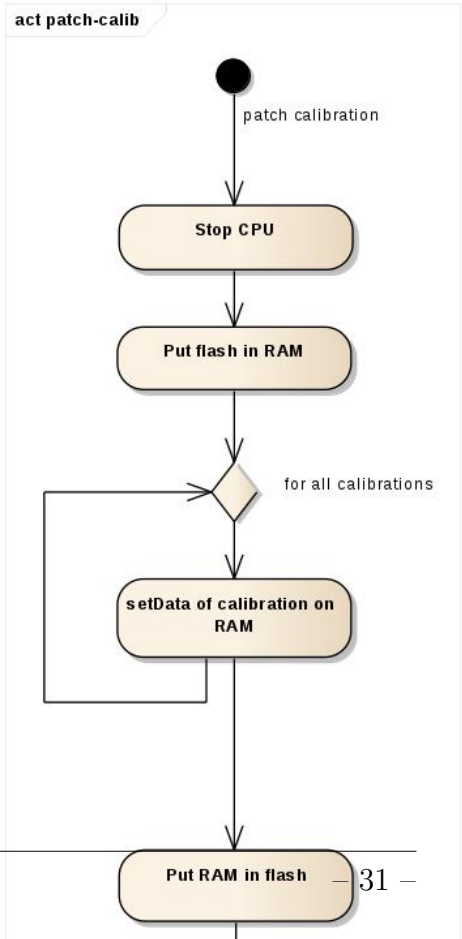
L'utilisateur à la possibilité d'écrire plusieurs scénarios dans un même test, avec cette même syntaxe.

5.2.3 Conception de la solution

Nous avons vu dans la section **??** que la fonctionnalité était répartie en deux grandes parties, le parser et l'exécution. Ces deux parties peuvent également s'apparenter à deux modules bien distincts de *GreenT* : le client, pour le parser, et le serveur Debugger, pour l'exécution. Dans la suite de ce document, nous feront abstraction de l'utilisation de Thrift pour la communication réseau, celle-ci étant généré comme expliqué section TODO SECTION TOOLS.

Le serveur T32 : Exécution

Comme expliqué section TODO REF, le serveur du debugger utilise l'outil Trace32. C'est cet outil qui nous permet d'interagir avec le calculateur embarqué. Cet outil possède fonctionne principalement en utilisant des scripts batch.
Depuis l'interface de cet outil, il est possible de patcher une calibration comme le montre la figure TODO test.



Le script existant Ce script a été pensé pour le mode graphique. L'utilisateur appelle le script, modifie les valeurs des calibrations voulues via l'interface , et cliquait sur « patch calibration ». À ce moment là, le patch en flash s'effectuait.

Le nouveau script Le nouveau script va effectuer les même actions que le script existant, mais sans aucune attente d'actions utilisateurs. Pour cela, plutôt que d'attendre que l'utilisateur modifies les calibrations, nous aurons une liste calibration à modifier.

Afin de modifier une calibration, présente en flash, un certain nombre d'actions sont nécessaires. Ces actions sont montrés dans

le diagramme d'activité figure 2.2.

Le service du serveur va ainsi appeler le nouveau script de patch, le code Java concerné sera donc minime. Le prototype de la méthode est la suivante :

```
public void patchCalibs(Map<String, Long> calibs);
```

Le client : Parser & générateur

Afin de gérer le parsing et la génération de notre patch calib, j'ai du créer une nouvelle règle de grammaire comme montré section **??**.

Avec cette nouvelle grammaire, j'ai forcé l'utilisateur à ne modifier des calibrations que si l'ECU est éteint, cette action n'étant possible que dans une `patch section`. Une fois la liste de calibration à patchée parsée, il m'a suffit de générer l'instruction d'appel du Debugger.

```
Map<String, Long> calibrations = new HashMap<String, Long>();
calibrations.put("calib1", 0x0);
calibrations.put("calib2", 0x1);
dbg.patchCalibs(calibrations);
// Stimulation instructions
```

Cependant, afin de répondre à un besoin de l'équipe cliente, j'ai ajouté une sauvegarde de la valeur des calibrations avant le premier scénario, et une restauration à la fin du dernier scenario. Ceci pour garantir une bonne cohérence, en effet, si l'utilisateur modifie une calibration, celle-ci doit être à la bonne valeur pour les tests suivants. Cette restauration se fait de façon implicite afin de limiter le travail des testeurs, qui pourrait être source d'erreur.

5.2.4 Les difficultés rencontrées

La robustesse

Le batch

5.3 Les « tableaux calibrables »

Certaines variables côtés ECU sont des variables de types tableau, l'utilisateur peut actuellement y accéder via la syntaxe `var[indice]`.

Le but de mon développement, était d'améliorer ce système. Actuellement, seul un indice en dur peut être renseigné à cette variable, l'idée est de pouvoir renseignée un indice en dur, mais aussi une calibration.

5.3.1 Expression du besoin

Cette fonctionnalité permettrai d'avoir des calibrations d'un projet à l'autre, et de pouvoir réutiliser le *Walkthrough* en ne changeant que les calibrations.
Un autre intérêt de cette fonctionnalité, est de pouvoir recopier les spécifications, celles-ci étant renseignée via des calibrations.

Avec cette fonctionnalité, l'utilisateur pourra donc avoir des fichiers *Walkthrough* génériques et réutilisable entre chaque versions d'un même projet. Cela limiterai également le risque d'erreur dûe à une mauvaise recopie de la spécification. En effet, avant cette fonctionnalité, l'utilisateur devait regarder la spécification, aller voir le contenu de la calibration, et la mettre en dur dans le

test. Cela peut provoquer des erreurs, d'une part lors de la recopie de la valeur, mais également si la valeur change à la version suivante et que l'utilisateur ne pense pas à faire suivre son document.

Les cas d'utilisation

Les limitations de la fonctionalité

5.4 La maintenance

Comme expliqué plus haut, j'ai développé deux fonctionnalités durant ce stage. Cependant, en parallèle de ce développement j'ai également corrigés différents bogues, ou améliorer différentes partie de la plateforme.

Ayant conçu cette plateforme lors de mon stage de fin de Licence, je connais l'intégralité de la plateforme. C'est ainsi que j'ai pu détecter et corriger un certains nombres de problèmes. Ces bogues ou ces améliorations ont été identifiées de trois façons différentes :

- Durant mon développement, il m'est arrivé de trouver du code incohérent ou bouchonné
- Lors d'exécutions de la plateforme sur différentes versions du projet client
- En regardant les différents tickets ouverts et devant être résolus

5.4.1 Corrections

J'ai corrigé quelques problèmes trouvés sur la plateforme, principalement venant du client *GreenT*. Ceci étant dû à notre choix d'architecture : des serveurs les plus légers et un client effectuant le maximum d'actions.

Stockage des erreurs d'exécutions

Le problème En cas d'erreur durant une exécution, si un serveur ne répond plus, si une variable est non trouvée, ... Une exception est levée. À ce moment là, *GreenT* doit attraper l'exception et la stocker en base de données pour pouvoir afficher ensuite le message à tous les tests du bundle, la génération des rapports ne pouvant pas se faire.

La solution Le stockage du message d'erreur n'était pas fait, ainsi que la requête SQL permettant d'obtenir les messages d'erreurs. Ces deux actions ont été corrigés, en lieu et place du rapport de test nous avons maintenant un message d'erreur clair.

Modification des variables Debugger

Le problème Lors d'un stimulus, il est possible de modifier une variable Debugger. Lors de la modification d'une variable, le Trace32 renvoyait toujours une exception, sans appliquer la modification.

La solution Après lecture de la documentation de Trace32, il s'est avéré que le problème venait simplement du serveur qui appliquait une commande syntaxiquement incorrecte. La modification de la commande a corrigé le problème.

Ordre d'exécutions des scénarios

Le problème Un Test peut contenir plusieurs scénarios, ceux-ci nous servent principalement pour le « patch-calib » comme expliqué section 2.2. Or, si nous utilisions plusieurs scénarios, ceux-ci étaient exécutés dans un ordre aléatoire : si l'utilisateur voulait effectuer des actions dans un ordre donnée, ce n'était pas possible.

La solution Le problème avait deux parties : d'une part, l'exécution des scénarios dans un ordre donné, d'autre part, spécifier un ordre à chacun de nos scénarios. En effet, tout d'abord, j'ai nommé les scénarios de sortes qu'ils soient classé par ordre alphabétique. Ensuite, il a fallu spécifier à la plateforme d'exécuter les différents scénarios dans un ordre alphabétique, pour cela il a suffit d'utiliser une collection Java effectuant cette action, la `TreeMap`.

Reset ECU à l'exécution

Le problème Lors de l'exécution de notre plateforme sur la dernière version du projet client, nous avions systématiquement un *reset* ECU. C'est-à-dire que notre ECU s'arrêtait et ne redémarrait pas pour une raison inconnue.

La solution Le problème ne venait pas directement de la plateforme, mais d'une mauvais configuration de notre part. En effet, nous ne spécifions pas les bons fichiers du logiciel, celui-ci étant mal flashé, l'ECU refusait de démarrer.

Absence d'injection

Le problème Lorsque j'essayais de simuler un démarrage du moteur sur la dernière version du logiciel, aucune injection ne se faisait : après le starter, le moteur retournait à zéro tour.

La solution Après s'être renseigné auprès de personnes compétentes, il s'est avéré que cela venait d'un nouveau fichier à flasher dont nous n'avions pas connaissance. Un fichier contenant des calibrations permettant le démarrage du moteur sur table. Ce fichier n'ayant pas été pris en compte durant la conception, j'ai ajouté un nouveau paramètre au fichier de configuration permettant de renseigner des fichiers à flasher additionnels.

5.4.2 Améliorations

Exécution différée

Le besoin Lors de mon développement, j'ai eu souvent des problèmes pour réservés des tables de tests. Celles-ci étant régulièrement prise par les équipes projets.

La solution Afin de ne pas bloquer de tables, et de ne pas bloquer notre travail en raison de l'absence de celles-ci, une solution nous est venue : la possibilité de lancer l'outil durant la nuit. En effet, actuellement une exécution dure environ 45 minutes, après laquelle nous pouvons analyser les rapports et voir les problèmes qui nous sont retournés. Ainsi, j'ai ajouté un nouveau paramètre à l'application permettant de spécifier l'heure à laquelle la génération des `.jar`, la compilation, l'exécution et la génération des rapports va se faire. On peut maintenant lancer une exécution le soir et observer les résultats le lendemain matin.

Passage à Java 8

Le besoin La plateforme fonctionnait sous Java 6. Ainsi, nous allions mettre en production une plateforme déjà obsolète à sa sortie. De plus, les deux versions suivantes de Java propose un certain nombre de fonctionnalités aidant au développement, comme des simplifications d'écriture en Java 7 (*Multi-Catch*, Inférence de type, `switch` sur les strings, ...) ou les lambdas-calculs en Java 8. Enfin, dans un future proche nous aurons besoin d'une interface pour *GreenT*, `JavaFx` serait une bonne solutions, mais celle-ci nécessite Java 8.

La solution Avant de passer à Java 8, il a d'abord fallut vérifier qu'aucune incompatibilité avec les bibliothèques que nous utilisons n'allait apparaître. Ensuite, il était nécessaire de télécharger un compilateur ainsi qu'une JVM, configurer les différents environnements et vérifier qu'une exécution se passait de la même manière que précédemment. Après ce succès, le passage à Java 8 a été concrétisé et permet à notre plateforme de rester moderne !

« Clean-Code »

Le besoin *GreenT* ayant deux ans, et ayant connue quatre développeurs différents, il est parfois nécessaire d'améliorer le code existant ou de le rendre plus lisible.

La solution Lorsqu'en développant mes fonctionnalités ou en corrigeant des bugs je tombais sur du code incompréhensible ou du « code mort », je modifiais celui-ci afin de corriger ces défauts. Ceci permet ainsi de garder un code toujours propre et facile à lire.

Affichage des logs

Le besoin La plateforme effectue beaucoup d'actions, allant du parsing jusqu'à la génération des rapports comme montré chapitre 1. Toutes ces actions doivent être tracés, aussi bien en temps réel, en regardant l'exécution, que plus tard en observant un fichier.

La solution Les logs fonctionnait déjà, en utilisant `log4j`, cependant celui-ci affichait beaucoup trop d'informations en temps réel, et n'affichait pas les informations les plus utiles. Ainsi, j'ai passé en revue la plateforme pour afficher les bonnes actions (Initialisation des bancs, stimulis effectués, affichage des exceptions, ...). Toutes les erreurs sont redirigé vers la sortie des erreurs (`stderr`), et seules les informations les plus importantes sont sur la sortie console (`stdout`). Tous les autres logs, qui peuvent être utile à la compréhension d'un problème et nous aider ne sont accessible que dans nos fichiers de logs. J'ai par ailleurs ajouté un « buffer tournant » permettant aux fichiers de logs de ne jamais dépasser une certaines taille (5Mio), afin de ne pas consommer trop d'espace.

6 Bilans

Après ce stage de quatre mois, il est temps de dresser un bilan, du point de vue de Continental afin de voir ce que mon travail leur a apporté, mais également en quoi ce stage a été bénéfique pour ma future carrière professionnelle.

6.1 Bilan pour Continental

Mon travail dans l'équipe de développement aura été intéressant pour l'entreprise, en partie grâce à ma connaissance de la plateforme suite à mon stage de licence. En effet, avoir contribué au projet l'année précédente sur la conception de celui-ci m'a permis de rapidement commencer le travail et de corriger des bugs répartis dans différents modules. De plus, revenir huit mois plus tard sur ce projet m'a permis d'appréhender le logiciel de manière plus globale et j'ai ainsi pu soulever des problèmes que nous n'avions pas vu lors de la conception.

Grâce à mes connaissances de l'architecture j'ai aidé Benjamin Guerin – le troisième membre de l'équipe arrivant sur le projet – j'ai ainsi pu lui donner des explications et des conseils, pendant que lui apportait un regard neuf à l'existant.

Comme présenté chapitre 2, mon travail aura été directement utile à l'équipe de développement et au groupe TAS. En effet, j'ai développé deux nouvelles fonctionnalités attendues par le client mais j'ai également corrigé des bugs. Ces corrections ainsi que les nouvelles fonctionnalités nous permettent maintenant d'exécuter les stimulations ainsi que l'analyse complète sur la dernière version du projet Ford : il est maintenant possible d'avoir les résultats de 79 tests en une heure, contre 51 tests lors de mon arrivée.

Le projet n'est pas terminé, et je n'ai pas pu effectuer toutes les fonctionnalités prévues tel que l'utilisation de GreenT avec d'autres fichiers d'entrées que le Walkthrough. Ceci est principalement dû à de mauvaises estimations, notamment en raison de la maintenance demandant plus de temps que prévu. Cependant, je vais continuer ce projet dès septembre en contrat de professionnalisation pendant un an et pourrai ainsi finaliser la plateforme.

6.2 Bilan personnel

Cette expérience en entreprise m'a beaucoup apporté, tout d'abord d'un point de vue technique, j'ai acquis de l'expérience en conception logicielle, grâce à toutes nos réunions où nous réfléchissions à la meilleure approche possible. De plus lors de problèmes, les propositions des autres m'ont permis d'avoir une autre vision du problème et une autre manière de le résoudre !

Contrairement à l'année précédente, j'ai pu travailler directement sur les tables de tests et pu ainsi découvrir des notions d'embarqués et d'automobile, et j'ai pu mieux appréhender le fonctionnement de l'ECU.

Mais j'ai aussi acquis des connaissances humaines avec notamment le travail en équipe, communiquer sur nos avancements, de manière écrite ou orale, et être capable de synthétiser ses propositions ou de réussir à manière claire et concise.

J'ai également eu le plaisir de revenir dans une multinationale, avec des collègues souhaitant toujours transmettre leurs connaissances et leur expérience, notamment dans le monde de l'automobile et de l'embarqué.

Un bilan très positif donc, qui m'a réconforté dans mon projet professionnel : ma continuation en M2 Développement Logiciel, en alternance chez Continental.

Annexes

Ci-après vous trouverez un certain nombre d'annexes qui pourront vous permettre de mieux comprendre l'étendue de mon travail et de la plateforme qui est en cours de développement.

Vous y trouverez ainsi un glossaire, des références, des exemples de rapport, de fichiers générés ainsi qu'une explication plus approfondie des équipements utilisés.

A Liste des codes sources

B Table des figures